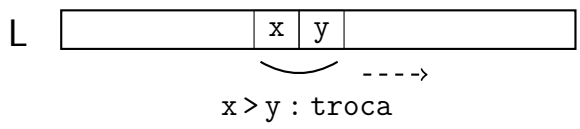
Mas, essa não é a única maneira de fazer as coisas.

Quer dizer, a gente também pode ordenar a lista fazendo varreduras.

O procedimento de *varredura* consiste em

→ *Percorrer a lista da esquerda para a direita comparando elementos consecutivos, e fazendo a troca quando eles estáo fora de ordem*



O efeito principal desse procedimento é levar o maior elemento de todos para a última posição — (*mas, quais são os seus efeitos colaterais?*)

Daí que, se a gente aplica varreduras sucessivas, a lista eventualmente fica completamente ordenada.

Para implementar essa ideia, o primeiro passo é escrever a função `Varredura( )`

```
func  Varredura ( L, inicio, fim )
 {
    Para  i <-- inicio  Até  fim - 1
     {
        Se ( L[i] > L[i+1] )
           Troca ( L, i, i+1 )
 }   }
```

Daí, a ordenação é feita botando o negócio para repetir

```
Para  j <-- N  Até  2
 {
    Varredura ( L, 1, j )
 }
```

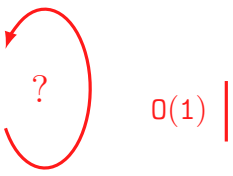
Esse algoritmo é conhecido a ordenação da bolha — (*porque?*)

E agora, nós vamos analisar o seu tempo de execução.

Análise

Para fazer isso, o primeiro passo é analisar a função `Varredura( )`.

E para isso, nós anotamos o seu código da seguinte maneira




```
func  Varredura ( L, inicio, fim )
 {
    Para  i <-- inicio  Até  fim - 1
     {
        Se ( L[i] > L[i+1] )
           Troca ( L, i, i+1 )
 }   }
```

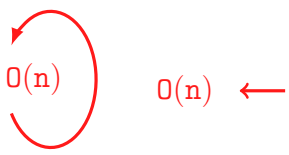
A observação fácil aqui é que o laço dá $O(n)$ voltas no pior caso — (*porque?*)

Daí que, no pior caso, a função executa em tempo

$$O(n) \cdot O(1) \qquad = \qquad O(n)$$

Certo.

Daí, sabendo disso, a gente anota o código principal assim




```
Para  j <-- 1  Até  N - 1
 {
    Varredura ( L, 0, N-j )
 }
```

Daí que, o tempo de execução da ordenação da bolha é

$$O(n) \cdot O(n) \qquad = \qquad O(n^2)$$

Exercício: Imagine que uma chamada à função `Varredura()` não realiza nenhuma troca.

Nesse caso, a ordenação da bolha já pode parar — (*porque?*)

Isso faz alguma diferença para o tempo de execução do algoritmo?

Bom, a resposta é: Sim e Não.

Porque?